# Intel® Platform Innovation Framework for EFI ACPI Specification

## *Draft for Review*

---

Version 0.91

August 8, 2006

# Revision History

| Revision | Revision History | Date |
|---|---|---|
| 0.9 | First public release. | 4/1/04 |
| 0.91 | Update for ACPI 3.0 | 8/8/06 |

# Contents

# Tables

# 1 Introduction

## Overview

This specification describes one design for supporting the *Advanced Configuration and Power Interface Specification* (hereafter referred to as the “ACPI specification”), revisions 1.0b and 2.0, in the Intel® Platform Innovation Framework for EFI (hereafter referred to as the "Framework"). The ACPI specification details system configuration and power management information.

This specification does the following:

- Describes the basic components of the Framework ACPI design, the ACPI support driver, and the ACPI platform driver
- Describes additional ACPI considerations for the Intel® Itanium® processor family
- Provides code definitions for the ACPI Support Protocol and other ACPI-related type definitions that are architecturally required by the *Intel® Platform Innovation Framework for EFI Architecture Specification*

See Industry Specifications in the Framework master help system for the URL for the ACPI specification.

## Scope

This specification provides a design for supporting the ACPI 1.0b and 2.0 specifications in a Framework environment.

## Conventions Used in This Document

This document uses the typographic and illustrative conventions described below.

### Data Structure Descriptions

Intel® processors based on 32-bit Intel® architecture (IA-32) are “little endian” machines. This distinction means that the low-order byte of a multibyte data item in memory is at the lowest address, while the high-order byte is at the highest address. Processors of the Intel® Itanium® processor family may be configured for both “little endian” and “big endian” operation. All implementations designed to conform to this specification will use “little endian” operation.

In some memory layout descriptions, certain fields are marked *reserved*. Software must initialize such fields to zero and ignore them when read. On an update operation, software must preserve any reserved field.

The data structures described in this document generally have the following format:

**STRUCTURE NAME:** The formal name of the data structure.

**Summary:** A brief description of the data structure.

**Prototype:** A "C-style" type declaration for the data structure.

**Parameters:** A brief description of each field in the data structure prototype.

**Description:** A description of the functionality provided by the data structure, including any limitations and caveats of which the caller should be aware.

**Related Definitions:** The type declarations and constants that are used only by this data structure.

### Protocol Descriptions

The protocols described in this document generally have the following format:

**Protocol Name:** The formal name of the protocol interface.

**Summary:** A brief description of the protocol interface.

**GUID:** The 128-bit Globally Unique Identifier (GUID) for the protocol interface.

**Protocol Interface Structure:** A "C-style" data structure definition containing the procedures and data fields produced by this protocol interface.

**Parameters:** A brief description of each field in the protocol interface structure.

**Description:** A description of the functionality provided by the interface, including any limitations and caveats of which the caller should be aware.

**Related Definitions:** The type declarations and constants that are used in the protocol interface structure or any of its procedures.

## Procedure Descriptions

The procedures described in this document generally have the following format:

**ProcedureName():** The formal name of the procedure.

**Summary:** A brief description of the procedure.

**Prototype:** A "C-style" procedure header defining the calling sequence.

**Parameters:** A brief description of each field in the procedure prototype.

**Description:** A description of the functionality provided by the interface, including any limitations and caveats of which the caller should be aware.

**Related Definitions:** The type declarations and constants that are used only by this procedure.

**Status Codes Returned:** A description of any codes returned by the interface. The procedure is required to implement any status codes listed in this table. Additional error codes may be returned, but they will not be tested by standard compliance tests, and any software that uses the procedure cannot depend on any of the extended error codes that an implementation may provide.

## Pseudo-Code Conventions

Pseudo code is presented to describe algorithms in a more concise form. None of the algorithms in this document are intended to be compiled directly. The code is presented at a level corresponding to the surrounding text.

In describing variables, a *list* is an unordered collection of homogeneous objects. A *queue* is an ordered list of homogeneous objects. Unless otherwise noted, the ordering is assumed to be First In First Out (FIFO).

Pseudo code is presented in a C-like format, using C conventions where appropriate. The coding style, particularly the indentation style, is used for readability and does not necessarily comply with an implementation of the *Extensible Firmware Interface Specification*.

## Typographic Conventions

This document uses the typographic and illustrative conventions described below:

| | |
|---|---|
| Plain text | The normal text typeface is used for the vast majority of the descriptive text in a specification. |
| Plain text (blue) | In the online help version of this specification, any plain text that is underlined and in blue indicates an active link to the cross-reference. Click on the word to follow the hyperlink. Note that these links are *not* active in the PDF of the specification. |

| | |
|---|---|
| **Bold** | In text, a **Bold** typeface identifies a processor register name. In other instances, a **Bold** typeface can be used as a running head within a paragraph. |
| *Italic* | In text, an *Italic* typeface can be used as emphasis to introduce a new term or to indicate a manual or specification name. |
| **BOLD Monospace** | Computer code, example code segments, and all prototype code segments use a **BOLD Monospace** typeface with a dark red color. These code listings normally appear in one or more separate paragraphs, though words or segments can also be embedded in a normal text paragraph. |
| **Bold Monospace** | In the online help version of this specification, words in a **Bold Monospace** typeface that is underlined and in blue indicate an active hyperlink to the code definition for that function or type definition. Click on the word to follow the hyperlink. Note that these links are *not* active in the PDF of the specification. Also, these inactive links in the PDF may instead have a **Bold Monospace** appearance that is underlined but in dark red. Again, these links are not active in the PDF of the specification. |
| *Italic Monospace* | In code or in text, words in *Italic Monospace* indicate placeholder names for variable information that must be supplied (i.e., arguments). |
| Plain Monospace | In code, words in a Plain Monospace typeface that is a dark red color but is not bold or italicized indicate pseudo code or example code. These code segments typically occur in one or more separate paragraphs. |

See the master Framework glossary in the Framework Interoperability and Component Specifications help system for definitions of terms and abbreviations that are used in this document or that might be useful in understanding the descriptions presented in this document.

See the master Framework references in the Interoperability and Component Specifications help system for a complete list of the additional documents and specifications that are required or suggested for interpreting the information presented in this document.

The Framework Interoperability and Component Specifications help system is available at the following URL:

http://www.intel.com/technology/framework/spec.htm

# 2
# Design Discussion

## ACPI Terms

The following terms are used throughout this specification or else are not widely used in other Framework specifications. See the master Glossary in the master help system for definitions of additional terms.

**ACPI**

Advanced Configuration and Power Interface.

**AML**

ACPI Machine Language.

**APIC**

Advanced Programmable Interrupt Controller.

**ASL**

ACPI Source Language.

**DSDT**

Differentiated System Description Table.

**EBDA**

Extended BIOS Data Area.

**FACS**

Firmware ACPI Control Structure.

**FADT**

Fixed ACPI Description Table.

**GPIO**

General Purpose Input/Output.

**MADT**

Multiple APIC Description Table.

**PMI**

Platform Management Interrupt.

**RSD_PTR**

Root System Description Pointer.

**RSDT**

Root System Description Table.

**SMI**

System Management Interrupt.

**SSDT**

Secondary System Description Table.

**S*x***

(Where *x* is a whole number from 1 to 5). Indicates the sleeping state of the system. See the ACPI industry specification for more information.

**XSDT**

Extended System Description Table.

## ACPI Overview

The Framework ACPI design separates common ACPI requirements from platform-specific ACPI requirements into different drivers to do the following:

- Maximize the reuse of support code
- Isolate the code that must be rewritten on a per-platform basis

To make an ACPI-compliant system, firmware must produce ACPI tables and report the ACPI memory usage to the operating system's EFI memory map reporting mechanisms (or via the INT 15 function E820h for legacy systems). The firmware may also be required to implement the following:

- Some processor-architecture-specific (System Management Interrupt [SMI]/Platform Management Interrupt [PMI]) code to support transitioning between legacy and ACPI modes
- A boot path to support S2/S3 resume

This specification describes a mechanism to accomplish these things. It divides the tasks into the following two components:

- ACPI support driver
- ACPI platform driver

The ACPI platform driver could actually be composed of several disparate drivers, but for the sake of this specification, it is assumed to be a single driver.

The ACPI support driver is generic and should be usable in widely varying platforms without modification. Its primary function is to control where in memory the ACPI tables are copied and to ensure that all ACPI tables have correct address links and checksums. The ACPI support driver also does the following:

- Creates generic tables such as the Root System Description Pointer (RSD_PTR), Root System Description Table (RSDT), and Extended System Description Table (XSDT) structures (see the ACPI industry specification for descriptions of these structures)
- Must publish the **`EFI_ACPI_SUPPORT_PROTOCOL`**, which is used by the ACPI platform driver

All platform differentiation with respect to ACPI occurs in the ACPI platform driver. The ACPI platform driver is responsible for the following:

- Producing all ACPI tables other than RSD_PTR, RSDT, and XSDT
- Installing any required SMI/PMI handlers
- Supporting any platform-specific sleep state requirements
- Generating the hardware signature

The ACPI platform driver may optionally publish interfaces to support sleep states.

# Rationale

## Rationale

The Framework strives to allow independent development and linking of components. While it would be nice to extend this idea to the ACPI name space and the ACPI Machine Language (AML) code, there are at least four reasons why this approach is not desirable:

- Name space collisions
- Compliance with the ACPI specification
- The complexity of dynamically generated AML
- Limitations of modular AML

The following topics discuss these reasons in detail.

## Name Space Collisions

Names in the ACPI name space are only four characters long. Each name must be unique in its scope. There is no way to guarantee that names will be unique if ACPI Machine Language (AML) is provided from different sources. The ACPI Source Language (ASL) code must be available to ensure there are no name collisions.

## ACPI Specification Compliance

ACPI is intended to replace nearly all BIOS interfaces for configuration, power management, and most things that are platform specific in nature. As of version 1.0b of the ACPI specification, many areas remain unclear. The specification is conspicuously quiet when it comes to defining roles and responsibilities for BIOS and operating system code. Because of these weaknesses, ACPI compliance is essentially the intersection of the following:

- ACPI specification
- Output of the Microsoft* ACPI Source Language (ASL) compiler
- Microsoft ACPI implementation requirements for the Windows* 98 and Windows 2000 operating systems

This intersection results in a constantly evolving set of requirements to which firmware writers need to conform. The Framework ACPI design allows for updating ACPI tables individually without requiring a full BIOS update.

If ACPI Machine Language (AML) was provided as precompiled modules from different sources, as Driver Execution Environment (DXE) drivers are, it would be impossible to make changes to meet the changing requirements for ACPI compliance.

### Complexity of Dynamically Generated AML

One possible solution to the problem would be to dynamically generate the name space from meta data that was provided in lieu of ACPI Machine Language (AML). The most immediate problem with this approach is that the code that is required to generate and parse an ACPI name space is large and complex. Because the Framework is intended to be system firmware, available space for code storage will be very limited.

### Limitations of Modular AML

Because the ACPI name space is inherently platform specific, it is nearly impossible to write correct ACPI Source Language (ASL) code without knowing the platform specifics.

For example: A silicon vendor could supply a chunk of ACPI Machine Language (AML) in the form of a Secondary System Description Table (SSDT), which could be loaded into the name space. However, the vendor could not possibly know how their chip was hooked up. What events were connected to which General Purpose Input/Output (GPIO)? The complexity of these problems likely makes this solution impossible.

## Requirements

This Framework ACPI design must meet the following requirements:

- All aspects of this design must be compliant with the following:
  - *Intel® Platform Innovation Framework for EFI Architecture Specification*, version 0.9
  - *Intel® Platform Innovation Framework for EFI Driver Execution Environment Core Interface Specification* (DXE CIS), version 0.9
  - ACPI 2.0 specification
  - ACPI 3.0 Specification
- The design must enable size efficiency, code reuse, and maintainability.
- It must be possible to access platform policy data from ACPI Machine Language (AML) code. For example, selections the user has made from a Setup application can be taken into account by control methods.

## ACPI Support Driver

### Introduction

The ACPI support driver is responsible for managing the memory map with regard to the ACPI tables. It also provides checksum services, creates some required tables (Root System Description Pointer [RSD_PTR], Root System Description Table [RSDT], Extended System Description Table [XSDT]), and updates Fixed ACPI Description Table (FADT) addresses.

The ACPI support driver also publishes the ACPI tables in the EFI Configuration Table.

### Dependency Resolution

The ACPI support driver requires only DXE Services. In other words, dispatch of the ACPI support driver is not dependent on any particular platform protocols.

### ACPI Support Protocol

See Code Definitions for the definition of the ACPI Support Protocol, which is the protocol definition that applies to the ACPI support driver.

## ACPI Platform Driver

### Introduction

The ACPI platform driver loads the ACPI tables. The driver relies on services provided by **EFI_ACPI_SUPPORT_PROTOCOL**, which is defined in Code Definitions.

ACPI requires a hardware signature to be generated by the Framework and written into the Firmware ACPI Control Structure (FACS) during each boot and each S4 resume. The hardware signature is intended to be a flag to the operating system that some hardware configuration has changed and the operating system must perform a cold boot. If the operating system is resuming from the S4 sleep state and it detects that the hardware signature has changed, it may abort the resume and do a cold boot instead. There is no meaning to the actual value of the signature, only whether it has changed from the previous boot. The ACPI platform driver is responsible for creating the hardware signature.

In addition, the ACPI platform driver must install any System Management Interrupt (SMI) or Platform Management Interrupt (PMI) support that is required by the ACPI specification. If the S3 sleep state is supported, the ACPI platform driver may also produce **EFI_BOOT_SCRIPT_SAVE_PROTOCOL**; see the *Intel® Platform Innovation Framework for EFI Boot Script Specification* for more details.

The ACPI platform driver may also complete additional platform-specific tasks. An example is updating the status of processors in the Multiple APIC Description Table (MADT) structure.

The driver is also responsible for determining which versions of the ACPI tables should be published by the ACPI support driver.

### Dependency Resolution

The ACPI platform driver requires the following to allocate memory for the tables and update tables accordingly:

- DXE Services
- **EFI_ACPI_SUPPORT_PROTOCOL**

The ACPI platform driver may require some chipset code to switch into and out of ACPI mode. Because this code is likely to run in System Management Mode (SMM) context, it must be linked (or loaded) in such a way as to allow it to exist in the SMM address range. Furthermore, it must not call out into non-SMM code.

The ACPI platform driver may have additional platform-specific requirements for obtaining information to determine table contents.

## Driver Execution

## Driver Execution

The ACPI platform driver is responsible for loading all tables that are required for a platform ACPI implementation using the **`EFI_ACPI_SUPPORT_PROTOCOL.SetAcpiTable()`** API.

Driver execution impacts the following:

- Table Selection
- SMI Implications

## Table Selection

A given implementation of the ACPI platform driver may have a variety of ACPI tables from which to choose when adding tables. It must be able to determine the correct set of tables to report. For example, whether or not to load a Secondary System Description Table (SSDT) for S3 support might depend on an environment variable that is used to toggle S3 support for the system.

## SMI Implications

IA-32 platforms may require System Management Interrupt (SMI) code for switching between legacy and ACPI operation modes and for handling global locks. This code will be registered with the SMI handler using the SMI protocol. Itanium-based systems may require similar Platform Management Interrupt (PMI) code.

## Platform Policy

In the context of the Framework, ACPI platform policy is defined to be any data that is to be consumed by either of the following:

- ACPI initialization code
- The ACPI Machine Language (AML) control methods themselves

For example, if a user disables a COM port using the Framework Setup utility, the _STA control method for that COM port could report it as "not present" to the operating system. This method would enable selections that were made in the Framework Setup utility to remain true in the context of the operating system. The policy data in this case is "COM port disabled."

Typically, this type of information has been stored in CMOS. Unless the chipset hardware provides an alternate access mechanism to read CMOS, there may be synchronization issues between the AML code and other software that is attempting to access CMOS at the same time.

Instead of using CMOS to report these types of details to AML code, the Framework will use a Secondary System Description Table (SSDT) to communicate policy data. The ACPI platform driver will load the correct SSDT. AML that needs access to policy data will reference this policy data using its name in the name space.

## System Sleep States

The table below describes the ACPI system sleep states that are supported in the Framework ACPI design.

**Table 2-1. Supported ACPI System Sleep States**

| ACPI Sleep State | Supported? | Notes |
|---|---|---|
| S1 | Yes | Pre-sleep: The Framework supports _PTS as necessary. |
| | | Post-sleep:<br>• The Framework supports _WAK as necessary.<br>• Nothing is required from the Framework. Waking is completely under the control of the operating system because resume occurs in the operating system rather than the reset vector. |
| S2 | Yes | Same as S3. |
| S3 | Yes | S3 resume is a special boot path that causes the Pre-EFI Initialization (PEI) phase to take different actions compared to a normal boot. Information must be saved by DXE during a normal boot that is retrieved by PEI during an S3 resume boot. See the *Intel® Platform Innovation Framework for EFI S3 Resume Boot Path Specification* for details on implementing S3 support. |
| S4 | Yes | Pre-sleep: The Framework supports _PTS as necessary. |
| | | Post-sleep:<br>• The Framework calculates the hardware signature.<br>• Normal POST except the EFI boot manager is bypassed and the operating system loader is invoked directly.<br>• The Framework supports _WAK as necessary. |
| S4BIOS | No | Framework support for S4BIOS is not defined. |

# Considerations for the Intel® Itanium® Processor Family

## ACPI Compliance

Early versions of ACPI implementations for the Itanium processor family supported a specification called *IA64 Extensions to the ACPI specification*. This specification is obsolete and has been superceded by the ACPI 2.0 specification. Framework-compliant systems must not support the *IA64 Extensions to the ACPI specification* and instead must comply with the ACPI 2.0 specification.

## Operating System Implementations

## Operating System Implementations

There are several differences between the operating system implementations for the Itanium processor family, which are all EFI aware, and the operating system implementations for IA-32 processors, of which some are EFI aware and others are not. The differences that are encountered depend on whether the target operating system is EFI aware. Following are some of these differences, which are also described in the next topics:

- The mechanism for locating the RSD_PTR structure
- The mechanism for retrieving the memory map

## Locating the RSD_PTR Structure

Legacy (non-EFI-aware IA-32) operating systems locate the Root System Description Pointer (RSD_PTR) structure by scanning for the RSD_PTR signature in the Extended BIOS Data Area (EBDA) and BIOS areas between E0000h and FFFFF on 16-byte boundaries.

EFI-aware operating systems (all Itanium-based systems and EFI-aware IA-32) locate the RSD_PTR structure by looking up its physical address in the EFI System Table. The ACPI support driver is responsible for updating the EFI System Table.

## Retrieving the Memory Map

Non-EFI-aware operating systems use the INT 15h function E820h to retrieve the system memory map. This retrieval must be handled by the legacy operating system boot code.

EFI-aware operating systems use the EFI Boot Service **`GetMemoryMap()`** to retrieve the system memory map. EFI Boot Services will properly report this information with no additional actions being required of the ACPI support driver.

# 3
# Code Definitions

## Introduction

This section contains the basic definitions for the Framework ACPI design. The following protocol is defined in this section:

- **EFI_ACPI_SUPPORT_PROTOCOL**

This section also contains the definitions for additional data types and structures that are subordinate to the structures in which they are called. The following data type can be found in "Related Definitions" of the parent function definition:

- **EFI_ACPI_TABLE_VERSION**

## ACPI Support Protocol

### EFI_ACPI_SUPPORT_PROTOCOL

#### Summary

This protocol provides some basic services to support publishing ACPI system tables. The services handle many of the more mundane tasks that are required to publish a set of tables. The services will do the following:

- Generate common tables.
- Update the table links.
- Ensure that tables are properly aligned and use correct types of memory.
- Update checksum values and IDs.
- Complete the final installation of the tables.

#### GUID

```
#define EFI_ACPI_SUPPORT_GUID \
{ 0xdbff9d55, 0x89b7, 0x46da, 0xbd, 0xdf, 0x67, 0x7d, 0x3d, 0xc0,
0x24, 0x1d }
```

#### Protocol Interface Structure

```
typedef struct _EFI_ACPI_SUPPORT_PROTOCOL {
  EFI_ACPI_GET_ACPI_TABLE     GetAcpiTable;
  EFI_ACPI_SET_ACPI_TABLE     SetAcpiTable;
  EFI_ACPI_PUBLISH_TABLES     PublishTables;
} EFI_ACPI_SUPPORT_PROTOCOL;
```

## Parameters

*GetAcpiTable*

Returns a table specified by an index if it exists. See the **GetAcpiTable()** function description.

*SetAcpiTable*

Adds, removes, or updates ACPI tables. See the **SetAcpiTable()** function description.

*PublishTables*

Publishes the ACPI tables. See the **PublishTables()** function description.

## Description

The **EFI_ACPI_SUPPORT_PROTOCOL** is published by the ACPI support driver and contains all the platform-independent interfaces that are used by the ACPI platform driver. The ACPI support driver's primary responsibility is to provide support functionality for the ACPI platform driver.

The ACPI support driver must produce and maintain valid structures for the following:

- Root System Description Pointer (RSD_PTR)
- Root System Description Table (RSDT)
- Extended System Description Table (XSDT)

See the ACPI specification for descriptions of these structures. They must be allocated from memory of type **EfiACPIReclaimMemory**. At all times, they must contain up-to-date pointers to existing tables.

The ACPI support driver maintains a list of tables that belong to each version of ACPI that is supported. Each version will have RSD_PTR, RSDT, and XSDT structures that are created by the driver. The one exception is the "none" version, which can be used to add items that do not belong to one of the other versions. Tables can be added to one or more versions of ACPI.

The ACPI support driver adds the RSD_PTR address to the EFI System Table, with the GUID that is defined in the ACPI specification, when another module (typically the ACPI platform driver) calls the **EFI_ACPI_SUPPORT_PROTOCOL.PublishTables()**. The ACPI support driver adds the RSD_PTR address to the EFI System Table using the **PublishTables()** function.

### NOTE

*Legacy ACPI operating systems (non-EFI-aware IA-32) search for the RSD_PTR signature, which is located either in the Extended BIOS Data Area (EBDA) or between E0000 and FFFFF. Legacy operating system code may have to ensure that RSD_PTR is copied or moved to a proper location.*

All physical addresses that are specified in the Fixed ACPI Description Table (FADT) structure need to be updated to reflect the actual table locations. The FADT needs the address of the Firmware ACPI Control Structure (FACS) and Differentiated System Description Table (DSDT) updated in the ACPI 1.0 and ACPI 2.0 portion of the table. The driver will have to update these addresses as tables are added and removed.

## EFI_ACPI_SUPPORT_PROTOCOL.GetAcpiTable()

### Summary

Returns a requested ACPI table.

### Prototype

```
typedef
EFI_STATUS
EFI_BOOTSERVICE
(EFIAPI *EFI_ACPI_GET_ACPI_TABLE) (
  IN EFI_ACPI_SUPPORT_PROTOCOL     *This,
  IN INTN                          Index,
  OUT VOID                         **Table,
  OUT EFI_ACPI_TABLE_VERSION       *Version,
  OUT UINTN                        *Handle
  );
```

### Parameters

*This*

A pointer to the **EFI_ACPI_SUPPORT_PROTOCOL** instance.

*Index*

The zero-based index of the table to retrieve.

*Table*

Pointer for returning the table buffer.

*Version*

Updated with the ACPI versions to which this table belongs. Type **EFI_ACPI_TABLE_VERSION** is defined in "Related Definitions" below.

*Handle*

Pointer for identifying the table.

### Description

The **GetAcpiTable()** function returns a buffer containing the ACPI table associated with the *Index* that was input. The following structures are not considered elements in the list of ACPI tables:

- Root System Description Pointer (RSD_PTR)
- Root System Description Table (RSDT)
- Extended System Description Table (XSDT)

The **EFI_ACPI_SUPPORT_PROTOCOL** instance allocates a buffer for returning a copy of the table from EFI Boot Services memory, and this buffer must be freed by the caller. *Table* will not point to the actual copy of the table.

*Handle* is a **UINTN** that is used to identify the table for use with the **EFI_ACPI_SUPPORT_PROTOCOL.SetAcpiTable()** function. The meaning of the value is implementation specific and cannot be used as anything more than an identifier for the table.

*Version* is updated with a bit map containing all the versions of ACPI of which the table is a member.

## Related Definitions

```
//
// ACPI Version bit map definition:
//
// EFI_ACPI_VERSION_1_0b - ACPI Version 1.0b
// EFI_ACPI_VERSION_2_0 - ACPI Version 2.0
// EFI_ACPI_TABLE_VERSION_3_0 - ACPI Version 3.0
// EFI_ACPI_VERSION_NONE - No ACPI Versions. This might be used
//  to create memory-based operation regions or other information
//  that is not part of the ACPI "tree" but must still be found
//  in ACPI memory space and/or managed by the core ACPI driver.
//
// Note that EFI provides discrete GUIDs for each version of ACPI
// that is supported. It is expected that each EFI GUIDed
// version of ACPI will also have a corresponding bit map
// definition. This allows maintenance of separate ACPI trees
// for each distinctly different version of ACPI.
//


//*******************************************************
// EFI_ACPI_TABLE_VERSION
//*******************************************************
#define EFI_ACPI_TABLE_VERSION          UINT32

#define EFI_ACPI_TABLE_VERSION_NONE     (1 << 0)
#define EFI_ACPI_TABLE_VERSION_1_0B     (1 << 1)
#define EFI_ACPI_TABLE_VERSION_2_0      (1 << 2)
#define EFI_ACPI_TABLE_VERSION_3_0      (1 << 3)
```

## Status Codes Returned

| | |
|---|---|
| EFI_SUCCESS | The function completed successfully. |
| EFI_NOT_FOUND | The requested index is too large and a table was not found. |

# EFI_ACPI_SUPPORT_PROTOCOL.SetAcpiTable()

## Summary

Used to add, remove, or update ACPI tables.

## Prototype

```
typedef
EFI_STATUS
EFI_BOOTSERVICE
(EFIAPI *EFI_ACPI_SET_ACPI_TABLE) (
  IN EFI_ACPI_SUPPORT_PROTOCOL     *This,
  IN VOID                          *Table    OPTIONAL,
  IN BOOLEAN                       Checksum,
  IN EFI_ACPI_TABLE_VERSION        Version,
  IN OUT UINTN                     *Handle
  );
```

## Parameters

*This*

A pointer to the **EFI_ACPI_SUPPORT_PROTOCOL** instance.

*Table*

Pointer to the new table to add or update.

*Checksum*

If **TRUE**, indicates that the checksum should be calculated for this table.

*Version*

Indicates to which version(s) of ACPI the table should be added. Type **EFI_ACPI_TABLE_VERSION** is defined in **EFI_ACPI_SUPPORT_PROTOCOL.GetAcpiTable()**.

*Handle*

Pointer to the handle of the table to remove or update.

## Description

The **SetAcpiTable()** function adds, updates, or removes ACPI tables. If **Handle* is zero and *Table* is not **NULL**, the function will add the table to the ACPI "trees" that are specified by *Version*. If **Handle* is not zero and *Table* is not **NULL**, the table(s) indicated by *Handle* and *Version* will be updated with the new table. If **Handle* is not zero and *Table* is **NULL**, the table(s) identified by the handle and *Version* will be removed. If **Handle* is zero and *Table* is **NULL**, **EFI_INVALID_PARAMETER** will be returned.

All tables that are added must be copied to memory of type **EfiACPIReclaimMemory**, except for the Firmware ACPI Control Structure (FACS), which must be of type **EfiACPIMemoryNVS**. The FACS must also be aligned on a 64-byte address boundary.

Checksum values for tables and structures must be calculated and put in place in compliance with the ACPI specification. *Checksum* will indicate which tables need their checksums updated. Additionally, the ACPI support driver must update any tables that it modifies as tables are added and removed, specifically the following:

- Root System Description Pointer (RSD_PTR)
- Root System Description Table (RSDT)
- Extended System Description Table (XSDT)
- Fixed ACPI Description Table (FADT)

The *Version* parameter is used to determine in which ACPI version a given table should be included. The ACPI 1.0b, ACPI 2.0 and ACPI 3.0 versions are separate but may contain pointers to common tables. *Version* allows the caller to specify which ACPI version should be updated for a given table.

## Status Codes Returned

| | |
|---|---|
| EFI_SUCCESS | The function completed successfully. |
| EFI_INVALID_PARAMETER | **Handle* was zero and *Table* was **NULL**. |
| EFI_ABORTED | Could not complete the desired action. |

# EFI_ACPI_SUPPORT_PROTOCOL.PublishTables()

## Summary

Causes one or more versions of the ACPI tables to be published in the EFI system configuration tables.

## Prototype

```
typedef
EFI_STATUS
EFI_BOOTSERVICE
(EFIAPI *EFI_ACPI_PUBLISH_TABLES) (
  IN EFI_ACPI_SUPPORT_PROTOCOL     *This,
  IN EFI_ACPI_TABLE_VERSION        Version
  );
```

## Parameters

*This*

A pointer to the **EFI_ACPI_SUPPORT_PROTOCOL** instance.

*Version*

Indicates to which version(s) of ACPI that the table should be published. Type **EFI_ACPI_TABLE_VERSION** is defined in **EFI_ACPI_SUPPORT_PROTOCOL.GetAcpiTable()**.

## Description

The **PublishTables()** function installs the ACPI tables for the versions that are specified in *Version*. No tables are published for *Version* equal to **EFI_ACPI_VERSION_NONE**. Once published, tables will continue to be updated as tables are modified with **EFI_ACPI_SUPPORT_PROTOCOL.SetAcpiTable()**.

## Status Codes Returned

| | |
|---|---|
| EFI_SUCCESS | The function completed successfully. |
| EFI_ABORTED | An error occurred and the function could not complete successfully. |